TP PROLOG

Algorithme d’unification Martelli-Montanari

Réalisé par :
BENKARRAD Alaa Eddine
HAFIANE Walid

SOMMAIRE

Question 1 : Algorithme de base

Premièrement, on discute les différents choix d'implantation des prédicats de bases de l'algorithme d'unification. Le but de cette première section, c'est la réalisation de prédicat **unifie(P)** où P est un système d'équations à résoudre représenté sous la forme d'une liste [S1 ?= T1,...,SN ?= TN]. Pour cela, on est besoin d'implémenter les prédicats regle, occur check et substitution qui sera détaillée juste après.

1. Regles

Le prédicat regle est une concrétisation des règles d'unification d'algorithmes d'unification. Elles sont données comme suit :

Rename $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est une variable

Simplify $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est une constante

Expand $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est composé et x n'apparaît pas dans t

Check $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow \bot$ si $x \neq t$ et x apparaît dans t

Orient $\{t \stackrel{?}{=} x\} \cup P'; S \rightsquigarrow \{x \stackrel{?}{=} t\} \cup P'; S$ si t n'est pas une variable

Decompose $\{f(s_1, \cdots, s_n) \stackrel{?}{=} f(t_1, \cdots, t_n)\} \cup P'; S \rightsquigarrow \{s_1 \stackrel{?}{=} t_1, \cdots, s_n \stackrel{?}{=} t_n\} \cup P'; S$

Clash $\{f(s_1, \cdots, s_n) \stackrel{?}{=} g(t_1, \cdots, t_m)\} \cup P'; S \rightsquigarrow \bot$ si $f \neq g$ ou $m \neq n$

On remarque bien que toutes ces règles sont composées d'une **condition** et une **opération** .
Le prédicat **regle(E, R)** permet seulement de déterminer la règle de transformation R qui s'applique à l'équation E, et non pas faire des opérations de transformation sur le système. Par conséquent, on s'interesse qu'aux conditions des règles, donc on a implanté le prédicat regle comme suit :

regle(X?= T, rename) si X et T sont des variables.
regle(X?= T, simplify) si X est une variable et T une constante.
regle(X?= T, expand) si X est une variable et X n'apparait pas dans t.
regle(X?= T, check) si X est une variable et apparaît dans t.
regle(T?= X, orient) si X est une variable et T ne l'est pas.
regle(T1 ?= T2, decompose) si T1 et T2 sont des termes complexes de meme arité et meme nom.
regle(T1?=T2, clash) si T1 et T2 sont des termes complexes de different nom ou arité.

Pour écrire ces conditions, on a utilisé les prédicats prédéfinis suivants :

var(X) : retourne vrai si X une variable, faux sinon.
non-var(X) : retourne vrai si X n'est pas une variable, faux sinon.
compound(X) : retourne vrai si X est un terme complexe, faux sinon.
functor(X, nom, arité) : retourne l'arité et le nom de terme.

On a aussi défini quelques prédicats pour simplifier la tâche.
same(T1, T2) : retourne vrai si les 2 termes complexes T1 et T2 ont le même nom et la même arité.
Occur check(X, T) : retourne vrai si X apparaît dans T, faux sinon . Ce prédicat sera décri dans la prochaine sous-section.

Après l'implantation des règles, on a jugé utile d'ajouter d'autres règles de simplification pour couvrir tous les cas possibles. Les règles sont les suivantes.

regle(X? = T, clean) si X et T sont le même terme.
Regle(T1? = T2, conste) si T1 est une constante et T2 un terme complexe, ou T2 une constante et T1 un terme complexe, en plus T1 et T2 ne sont pas les mêmes.

2. Test d'occurence

Le teste d'occurrence joue un rôle important dans l'efficacité d'algorithmes d'unification. Pour cela l'implantation de prédicat **d'occur check(X, T)** est non négligeable. On peut considérer ses étapes comme suit :

- si le terme T est une constante : faux.
- sinon si T est une variable : renvoi vrai si X est la meme variable, faux sinon.
- sinon T est un terme complexe : on execute l'occur_check pour tous les arguments de T, renvoi vrai si un des arguments renvoi vrai, faux sinon.

3. Reduction

Dans cette section, on explique l'implantation de prédicat de réduction **reduit(R, E, P, Q)** qui permet de transformer le système d'équations P en le système d'équations Q par application de la règle de transformation R à l'équation E., mais avant de l'entamer, on doit définir les prédicats nécessaires pour son fonctionnement.

substitution(X?=T, P, Q): c'est l'implantation de processus de substitution en logique, qui permet de substituer le **terme X** par le **terme T** dans P et renvoi le résultat dans Q . ce prédicat nous permet d'effectuer rename, simplify et expand.

subs_term(X, T, Term, TRes) : ce prédicat est utilisé dans substitution, contrairement à ce dernier sub fait des substitutions sur un seul terme et non pas sur un système.

subs_term_list(X,T, Lterm, Lres) : ce prédicat fait des substitutions sur une liste des termes Lterm et renvoi la nouvelle liste dans Les, il est utilisé pour les termes complexes.

orient(T1 ?= T2, T2 ?= T1) : Permets tout simplement de réorienter les deux termes d'unification.

decomp(T1 ?= T2, Q) : Décompose l'équation T1 ? = T2 et renvoi un système dans Q.

Le prédicat de réduction était très simple à mettre en œuvre, selon la regle applicable, il fait appel au prédicat concerné.
reduit(clean, E, P, P) : dans ce cas on ne fait rien.
reduit(rename, E, P, Q) : fait appel a substitution(E,P,Q),
reduit(simplify, E, P, Q) : fait appel a substitution(E,P,Q),
reduit(expand, E, P, Q) : fait appel a substitution(E,P,Q),
reduit(check, E, P, ['⊥']) : renvoi un systeme contradictoire.
reduit(decompose, E, P, Q) : fait appel a decomp(E, Qtemp) et renvoi le system [Qtemp, P] dans Q.
reduit(orient, E, P, Q) : fait appel a orient(E,Qtemp)
reduit(clash, E, P, ['⊥']) : renvoi un systeme contradictoire.
reduit(const, E, P, ['⊥']) : renvoi un systeme contradictoire.
À chaque exécution de prédicat réduit, on affiche le system et la regle courante.

4. Unification

Dans ce qui précède, on a défini tous les prédicats nécessaires à l'implantation de prédicat d'unification **unifie(P)**. Ce dernier peut être défini tous simplement comme suit :

- On extrait une équation E de P, et on la passe en argument de l'appel de **réduit(R, E, P, Q)** qui nous renvoie le nouveau système dans Q.
- On repete l'opération précédente par unifie(Q) sur le nouveau système. On s'arrête quand le système est vide ou contradictoire (contient le symbole ⊥).

Question 2 : Strategies

Dans cette section, on explique les différentes stratégies implantées. Au début, on a jugé important de citer que pour chaque element E d'un système donné P, l'algorithme s'unifie avec une et une seule regle.

1. Choix pemier

Cette stratégie est la stratégie par défaut, elle se justifie par le fait d'extraits, a chaque étape, la première regle de système et applique à cette dernière la 1re regle applicable. On ne détaille pas sur cette stratégie car comme mentionné, c'est l'implémentation par défaut.

2. Choix pondéré

De l'autre côté, une stratégie intéressante qui consiste à donner des poids aux règles. Ensuite, on applique la regle (doit être applicable) qui possède le plus grand poids en premier. Cette approche est plus efficace et plus optimisée. Pour réaliser cette approche on a défini, tous d'abord, la base des faits, poids des règles :

```
poid(clean, 6).
poid(check, 5).
poid(const, 0).
poid(orient, 3).
poid(expand, 1).
poid(simplify, 4).
poid(clash, 5).
poid(decompose, 2).
poid(rename, 4).
```

Ensuite, on definit les deux prédicats :
get_eq_rule(Poid, P, E, R, Q) : ce prédicat renvoie la regle R de poids Poids, qui s'applique sur l'element E de P, et Q contient le reste du système P (Q = P-{E}).
poid_max(P, Max) : ce prédicat parcour tous le système P, et renvoie le plus grand poids des règles applicables.

Donc le prédicat de la stratégie choix_pondéré sera défini comme suit ;
choix_pondere([],[],_,_).
choix_pondere(P, Q, E, R) :- appler **poid_max(P, Max)** ensuite **get_eq_rule(Max, P, E, R, Q)**.

3. Choix randomisé

Afin de rendre les choses plus intéressantes, on a ajouté une stratégie **random** qui consiste à tirer un element **E** de **P** au **hasard**, puis appliquer une regle sur cet element.Cette approche est totalement randomisée, par conséquent, elle peut parfois, retourner un résultat semblant au résultat de l'une des deux stratégies précédent.

Question 3 : Trace

Cette section sera très courte, on présente la mise n'œuvre des prédicats de trace, sans rien dire, voici l'implantation de prédicat :

```
trace_unif(P) :  set_echo, unifie(P).
trace_unif(P,S) :  set_echo, unifie(P,S).

unif(P) :  clr_echo, unifie(P).
unif(P,S) :  clr_echo, unifie(P,S).
```

Implantation

La dernière section est consacrée à la trace d'exécution de notre implantation des différents tests. Mais juste avant, ont presente un listing des fichiers sources.

1. Listing

prerequis.pl : contient la définition d'opérateur et les prédicats d'affichage
regles.pl : contient le prédicat qui définit l'ensemble des règles.
global_sol.pl : contiennent les prédicats qui gèrent la variable globale de la solution.
substitution.pl : contient les prédicats de substitution.
reduction.pl : contient les prédicats nécessaires à la réduction.
strategies.pl : contiennent des prédicats de stratégies.
uni.pl : le fichier de l'unificateur, contient les prédicats d'unification.
cons_menu.pl : contient les prédicats d'affichage et de saisie de l'interface console.

main.pl : le programme principale qui contient la boucle.
README.txt : contient des instruction sur l'utilisation d'unificateur
trace_tests.txt : contient les traces d'exécutions de tous nos tests

2. Traces d'execution

Cette section représente qu'une partie des traces d'exécution pour plus de trace veuillez consulter le fichier **trace_tests.txt.**

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)]).
systeme : [f(_6,_8)?=f(g(_10),h(a)),_10?=f(_8)]
decompose : f(_6,_8)?=f(g(_10),h(a))
systeme : [_6?=g(_10),_8?=h(a),_10?=f(_8)]
expand : _6?=g(_10)
systeme : [_8?=h(a),_10?=f(_8)]
expand : _8?=h(a)
systeme : [_10?=f(h(a))]
expand : _10?=f(h(a))

YES

X = g(f(h(a))),
Y = h(a),
Z = f(h(a)).
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], premier).
systeme : [f(_4900,_4902)?=f(g(_4906),h(a)),_4906?=f(_4900)]
decompose : f(_4900,_4902)?=f(g(_4906),h(a))
systeme : [_4900?=g(_4906),_4902?=h(a),_4906?=f(_4900)]
expand : _4900?=g(_4906)
systeme : [_4902?=h(a),_4906?=f(g(_4906))]
expand : _4902?=h(a)
systeme : [_4906?=f(g(_4906))]
check : _4906?=f(g(_4906))

NO
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], pondere).
```

```
systeme : [f(_6314,_6316)?=f(g(_6320),h(a)),_6320?=f(_6314)]
decompose : f(_6314,_6316)?=f(g(_6320),h(a))
systeme : [_6314?=g(_6320),_6316?=h(a),_6320?=f(_6314)]
expand : _6314?=g(_6320)
systeme : [_10?=f(g(_10)),_8?=h(a)]
check : _10?=f(g(_10))

NO
```

```
?- trace_unif([Z?=X, X?=f(a), Z?=f(b)]).
systeme : [_2224?=_2226,_2226?=f(a),_2224?=f(b)]
rename : _2224?=_2226
systeme : [_2224?=f(a),_2224?=f(b)]
expand : _2224?=f(a)
systeme : [f(a)?=f(b)]
decompose : f(a)?=f(b)
systeme : [a?=b]
const : a?=b

NO
```

```
?- trace_unif([f(X,X,X)?=f(f(Y,Y,Y),f(Y,Y,Z),a)]).
systeme : [f(_6,_6,_6)?=f(f(_8,_8,_8),f(_8,_8,_10),a)]
decompose : f(_6,_6,_6)?=f(f(_8,_8,_8),f(_8,_8,_10),a)
systeme : [_6?=f(_8,_8,_8),_6?=f(_8,_8,_10),_6?=a]
expand : _6?=f(_8,_8,_8)
systeme : [f(_8,_8,_8)?=f(_8,_8,_10),f(_8,_8,_8)?=a]
decompose : f(_8,_8,_8)?=f(_8,_8,_10)
systeme : [_8?=_8,_8?=_8,_8?=_10,f(_8,_8,_8)?=a]
clean : _8?=_8
systeme : [_8?=_8,_8?=_10,f(_8,_8,_8)?=a]
clean : _8?=_8
systeme : [_8?=_10,f(_8,_8,_8)?=a]
rename : _8?=_10
systeme : [f(_8,_8,_8)?=a]
const : f(_8,_8,_8)?=a

NO
```

```
?- trace_unif([f(X,X,X)?=f(f(Y,Y,Y),f(Y,Y,Z),a)], pondere).
systeme : [f(_12,_12,_12)?=f(f(_14,_14,_14),f(_14,_14,_16),a)]
decompose : f(_12,_12,_12)?=f(f(_14,_14,_14),f(_14,_14,_16),a)
systeme : [_12?=a,_12?=f(_14,_14,_14),_12?=f(_14,_14,_16)]
simplify : _12?=a
systeme : [a?=f(_14,_14,_14)]
const : a?=f(_14,_14,_14)

NO
```

```
?- trace_unif([f(X,X,X)?=f(f(Y,Y,Y),f(Y,Y,Z),a)], random).
systeme : [f(_1756,_1756,_1756)?=f(f(_1764,_1764,_1764),f(_1764,_1764,_1776),a)]
decompose : f(_1756,_1756,_1756)?=f(f(_1764,_1764,_1764),f(_1764,_1764,_1776),a)
systeme : [_1756?=f(_1764,_1764,_1764),_1756?=f(_1764,_1764,_1776),_1756?=a]
expand : _1756?=f(_1764,_1764,_1764)
systeme : [f(_1764,_1764,_1764)?=f(_1764,_1764,_1776),f(_1764,_1764,_1764)?=a]
decompose : f(_1764,_1764,_1764)?=f(_1764,_1764,_1776)
systeme : [f(_1764,_1764,_1764)?=a,_1764?=_1764,_1764?=_1764,_1764?=_1776]
const : f(_1764,_1764,_1764)?=a

NO
```

```
?- trace_unif([f(X,X,X)?=f(f(Y,Y,Y),f(Y,Y,Z),a)], random).
systeme : [f(_442,_442,_442)?=f(f(_450,_450,_450),f(_450,_450,_462),a)]
decompose : f(_442,_442,_442)?=f(f(_450,_450,_450),f(_450,_450,_462),a)
systeme : [_442?=f(_450,_450,_450),_442?=f(_450,_450,_462),_442?=a]
expand : _442?=f(_450,_450,_450)
systeme : [f(_450,_450,_450)?=a,f(_450,_450,_450)?=f(_450,_450,_462)]
const : f(_450,_450,_450)?=a

NO
```
